

Intel Threading Tools for Microsoft® Windows and Linux



Threading Tools Evolution for Intel Architectures

- Intel's KAP/Pro Tool Set Available Now For:
 - Microsoft® Windows
 - Linux
- Will be Supplanted by Intel's Threading Tools

Threading Tool Components

- Part of VTune™
- There are two basic threading tool components:
 - Thread Analyzer – measures parallel execution performance
 - Intel® Thread Checker – validates execution of application based on an input dataset

How Does the Thread Analyzer Work?

Let us look at programming an algorithm
and then proceed to optimize the
execution performance with the VTune™
Thread Analyzer



Algorithm – Sieve of Eratosthenes

- Eratosthenes of Cyrene lived around 200 B.C.
- The prime number sieve begins by writing down the integers between N and N^2
- Remove the composite numbers in stages
- First all multiples of 2 are removed; then all multiples of 3 are removed, and so on
- The process stops after sifting with the largest prime less than N

Prime Number Theorem

- The number of primes less than N is approximately $N/\ln N$
- Thus the primes are relatively dense in the integers

Example Where $N = 6$ and Therefore $N^2 = 36$

- Stage 0 (initially)

- 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36

- Stage 1 (multiples of 2 eliminated)

- 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36

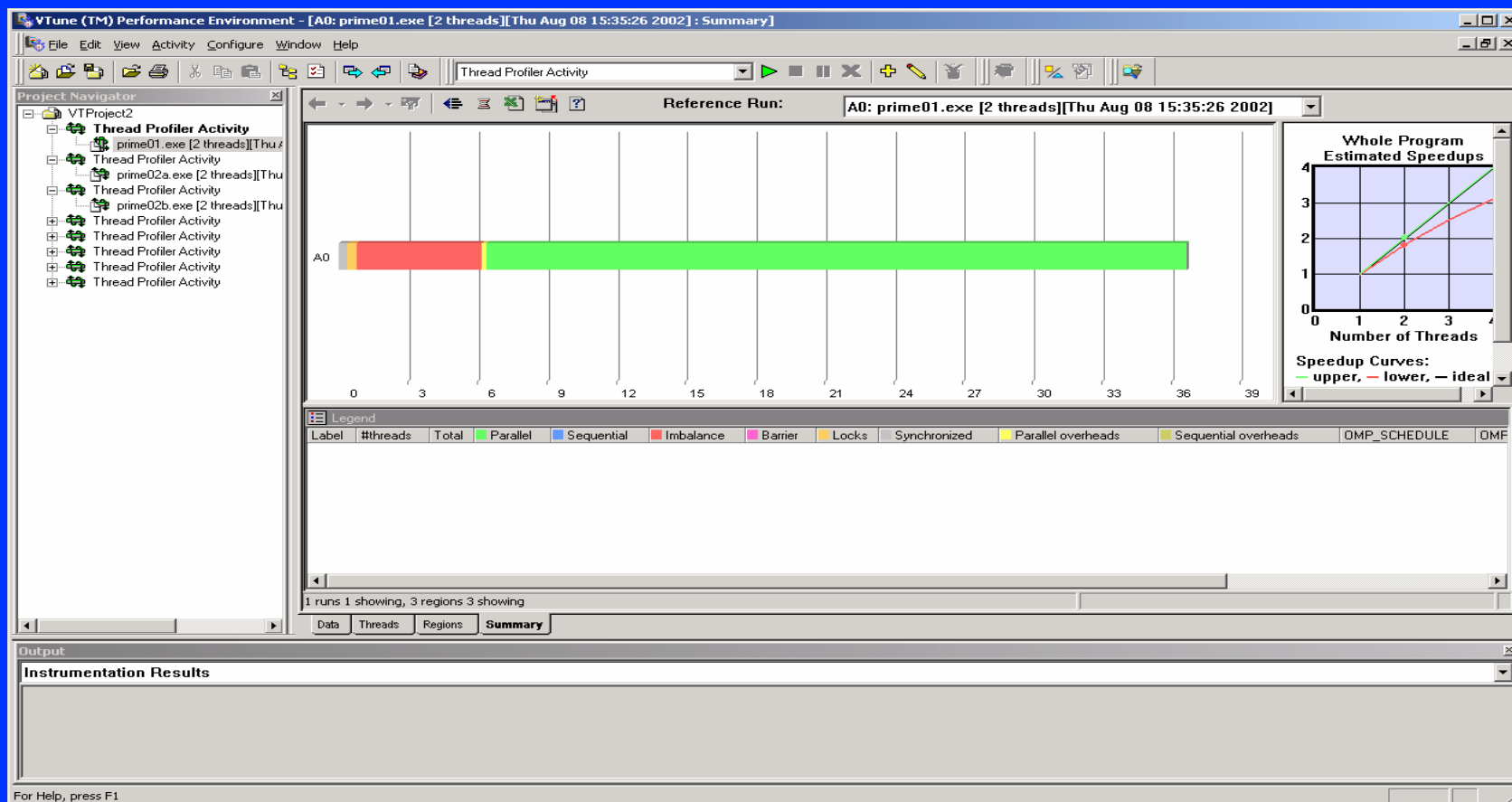
- Stage 3 (multiples of 3 eliminated)

- 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36

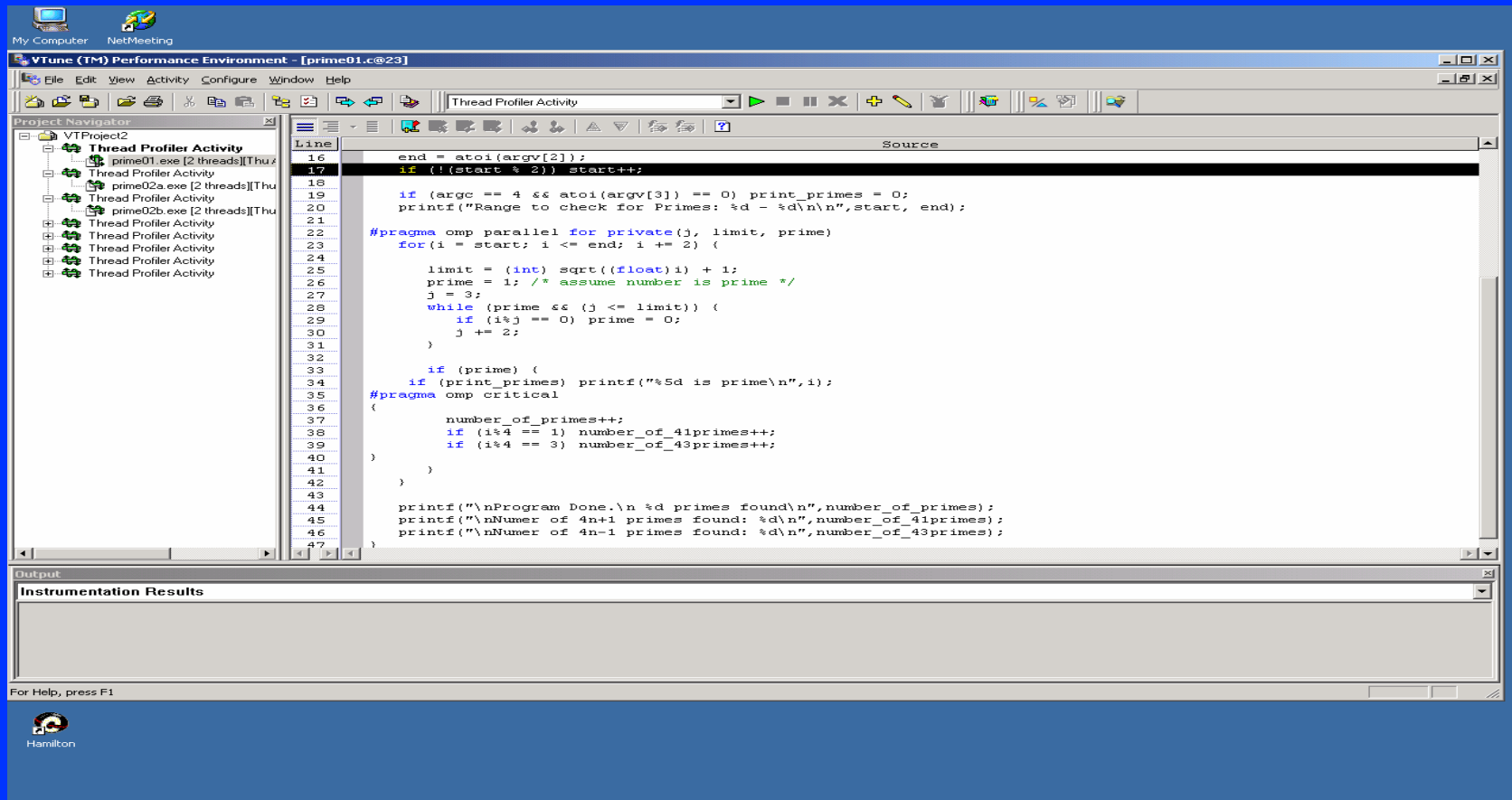
- Stage 4 (multiples of 5 eliminate)

- 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36

Threading Performance Analysis Panel



The red section of the histogram on previous slide signifies a load imbalance. Let us look at the corresponding source:



The screenshot displays the VTune Performance Environment interface. The main window shows the source code for a program that finds prime numbers. The code is in C and uses OpenMP for parallelization. The output window at the bottom shows the results of the program.

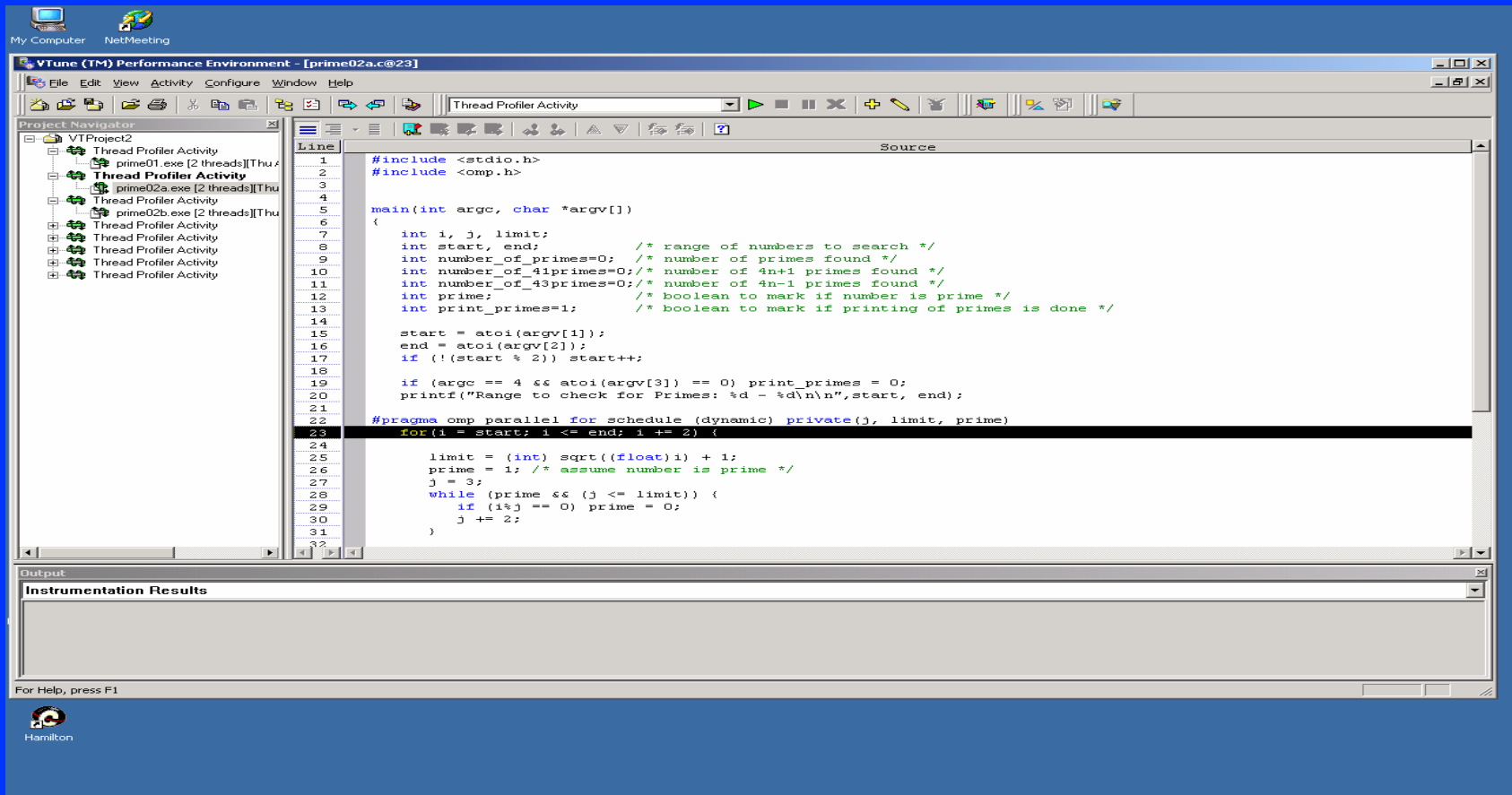
```
16  end = atoi(argv[2]);
17  if (!(start % 2)) start++;
18
19  if (argc == 4 && atoi(argv[3]) == 0) print_primes = 0;
20  printf("Range to check for Primes: %d - %d\n", start, end);
21
22  #pragma omp parallel for private(j, limit, prime)
23  for(i = start; i <= end; i += 2) {
24
25      limit = (int) sqrt((float)i) + 1;
26      prime = 1; /* assume number is prime */
27      j = 3;
28      while (prime && (j <= limit)) {
29          if (i%j == 0) prime = 0;
30          j += 2;
31      }
32
33      if (prime) {
34          if (print_primes) printf("%5d is prime\n", i);
35          #pragma omp critical
36          {
37              number_of_primes++;
38              if (i%4 == 1) number_of_41primes++;
39              if (i%4 == 3) number_of_43primes++;
40          }
41      }
42
43      printf("\nProgram Done.\n %d primes found\n", number_of_primes);
44      printf("\nNumber of 4n+1 primes found: %d\n", number_of_41primes);
45      printf("\nNumber of 4n-1 primes found: %d\n", number_of_43primes);
46
47  }
```

Output

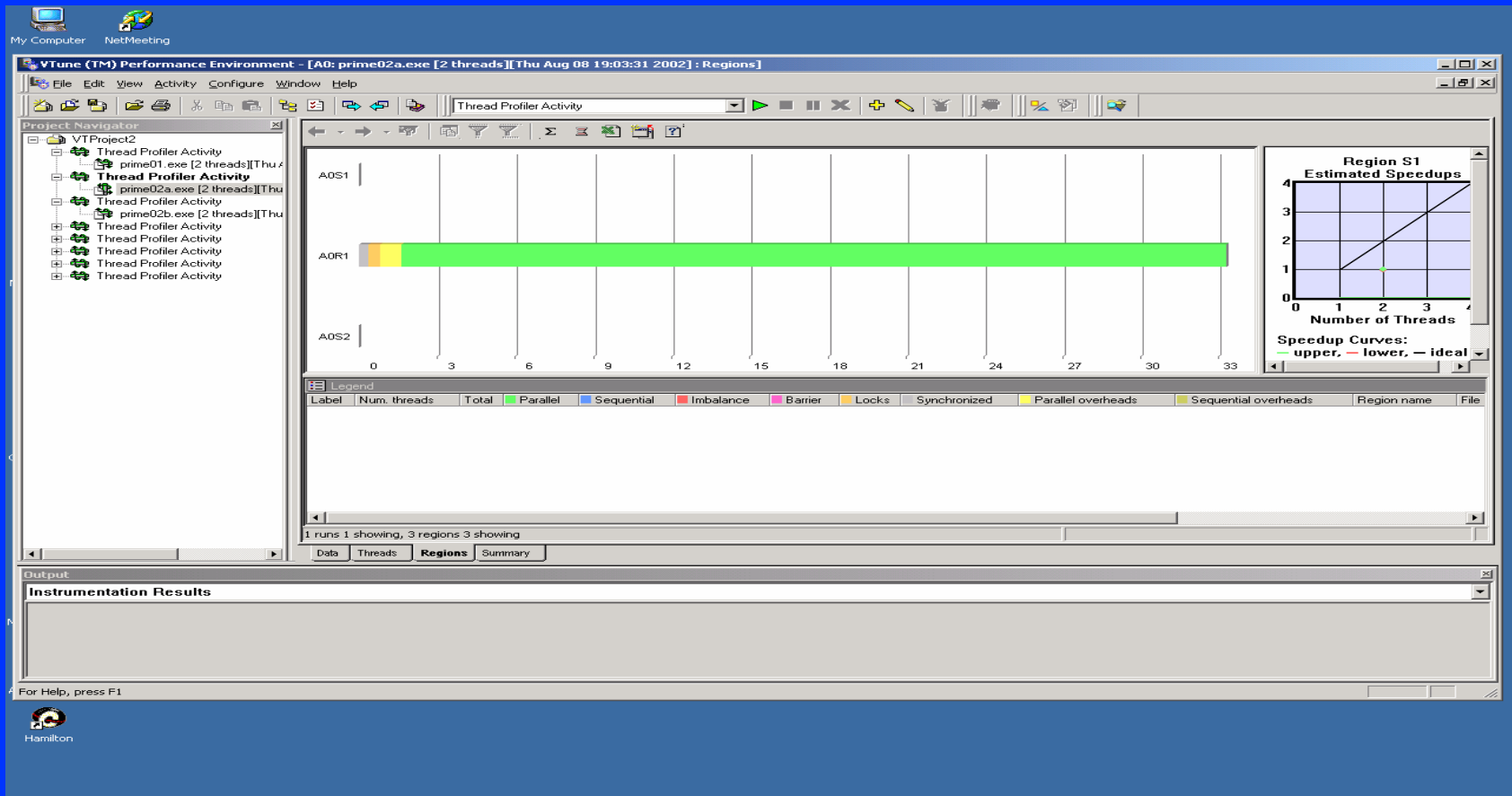
Instrumentation Results

For Help, press F1

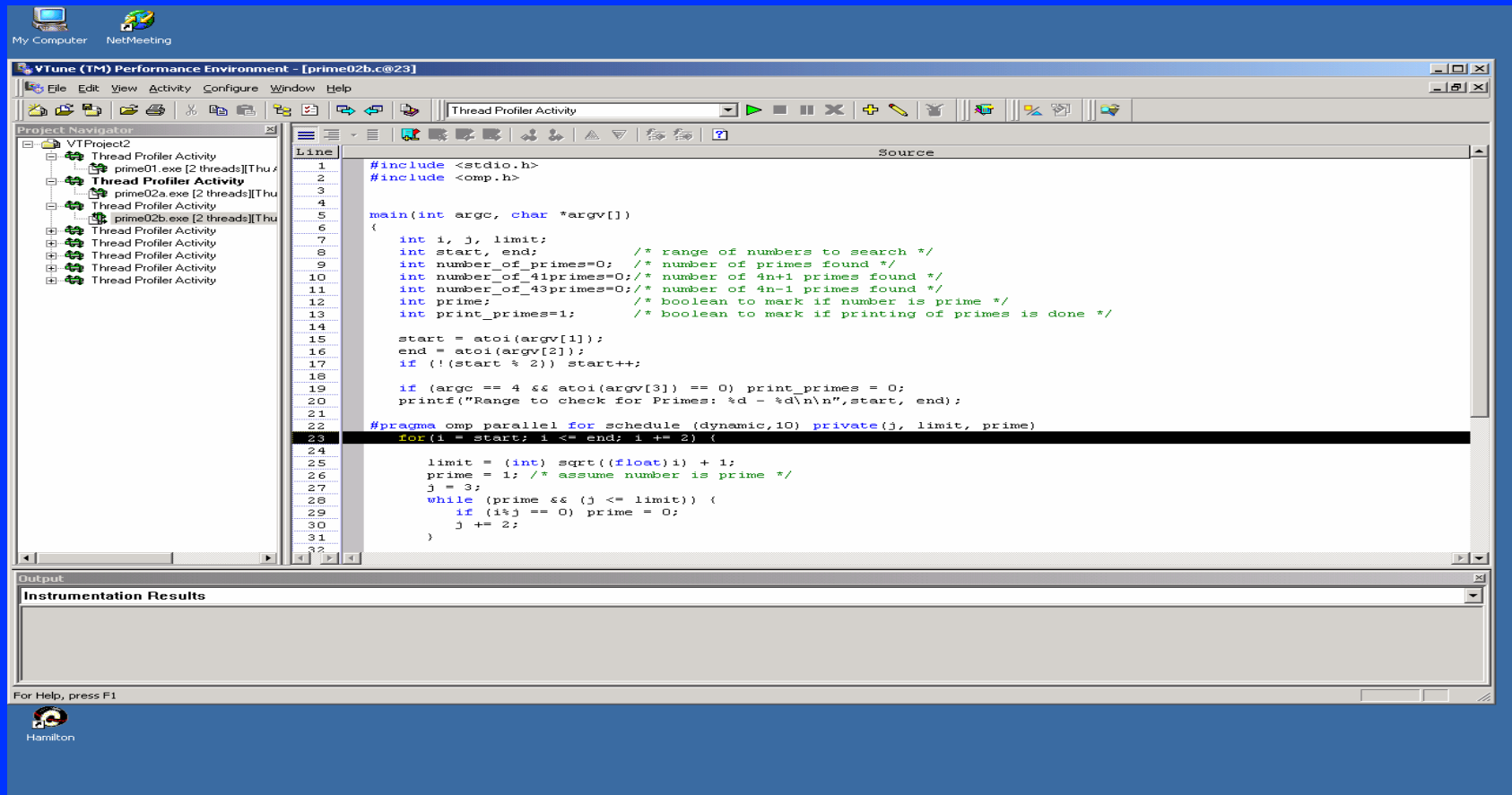
Alter the OpenMP Application by Adding Dynamic Scheduling



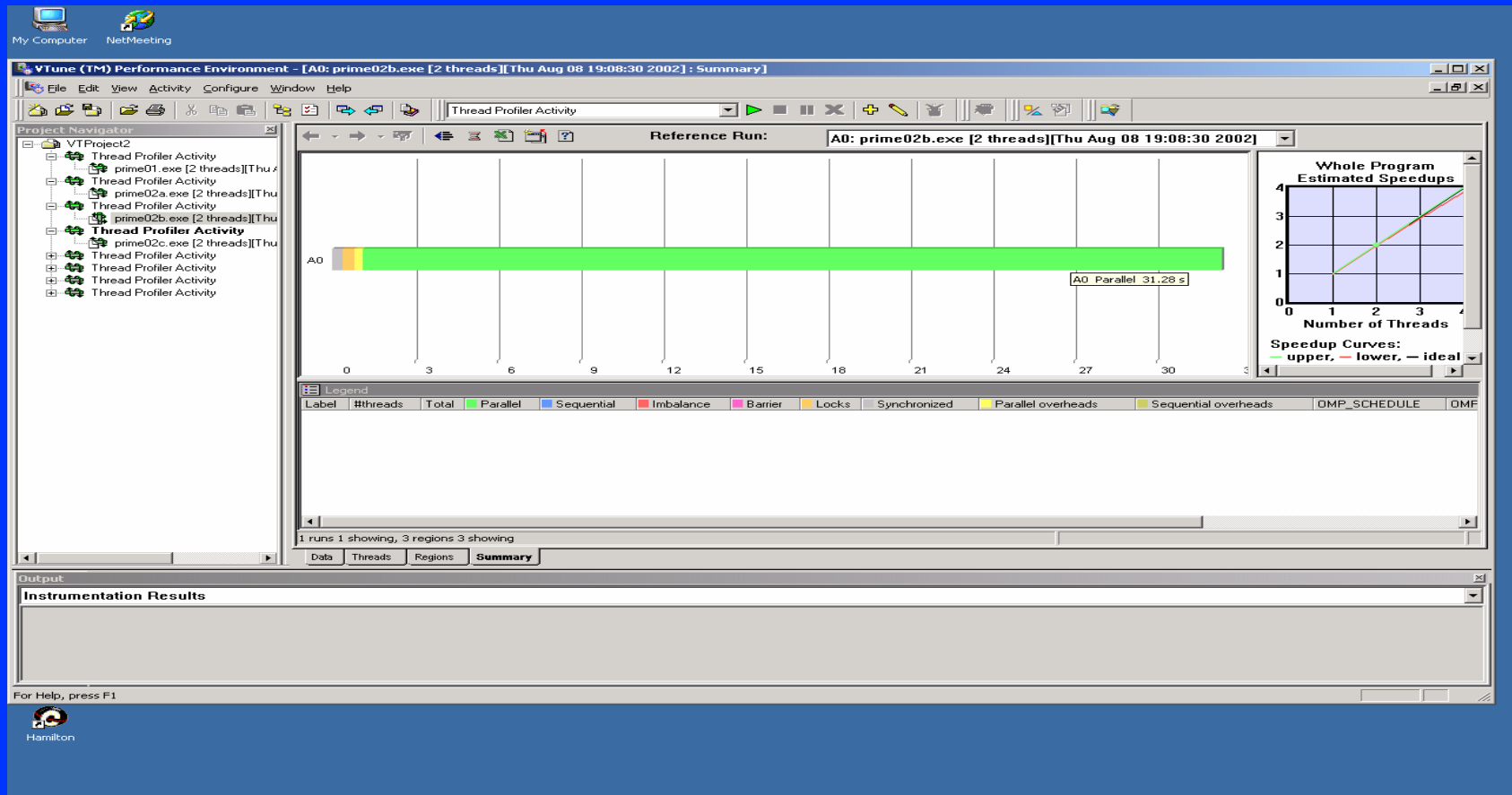
Histogram after Adjusting for Dynamic Scheduling



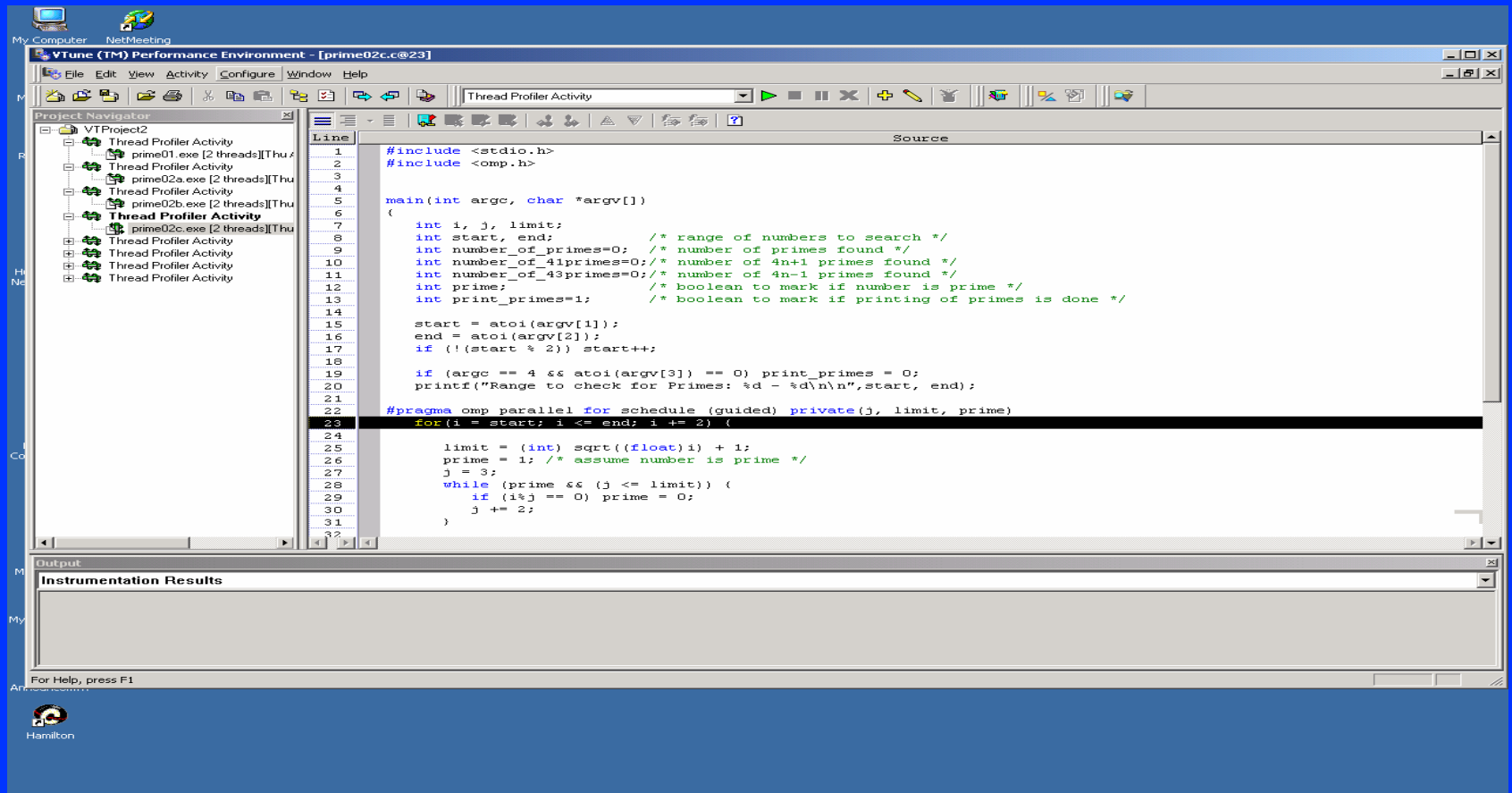
Adjust the Dynamic Scheduling with a Chunksize of 10



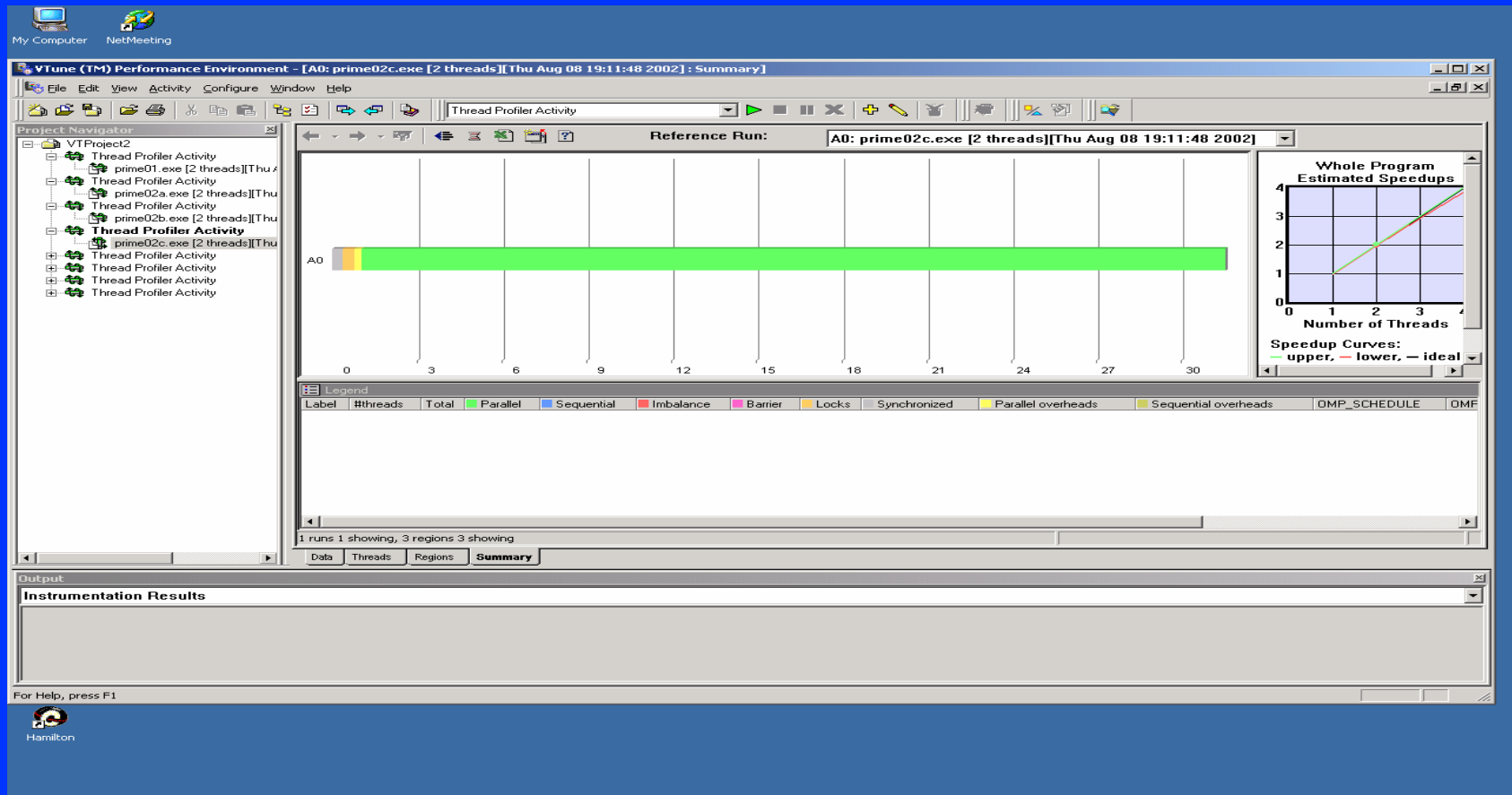
Notice the Improvement in Execution Time



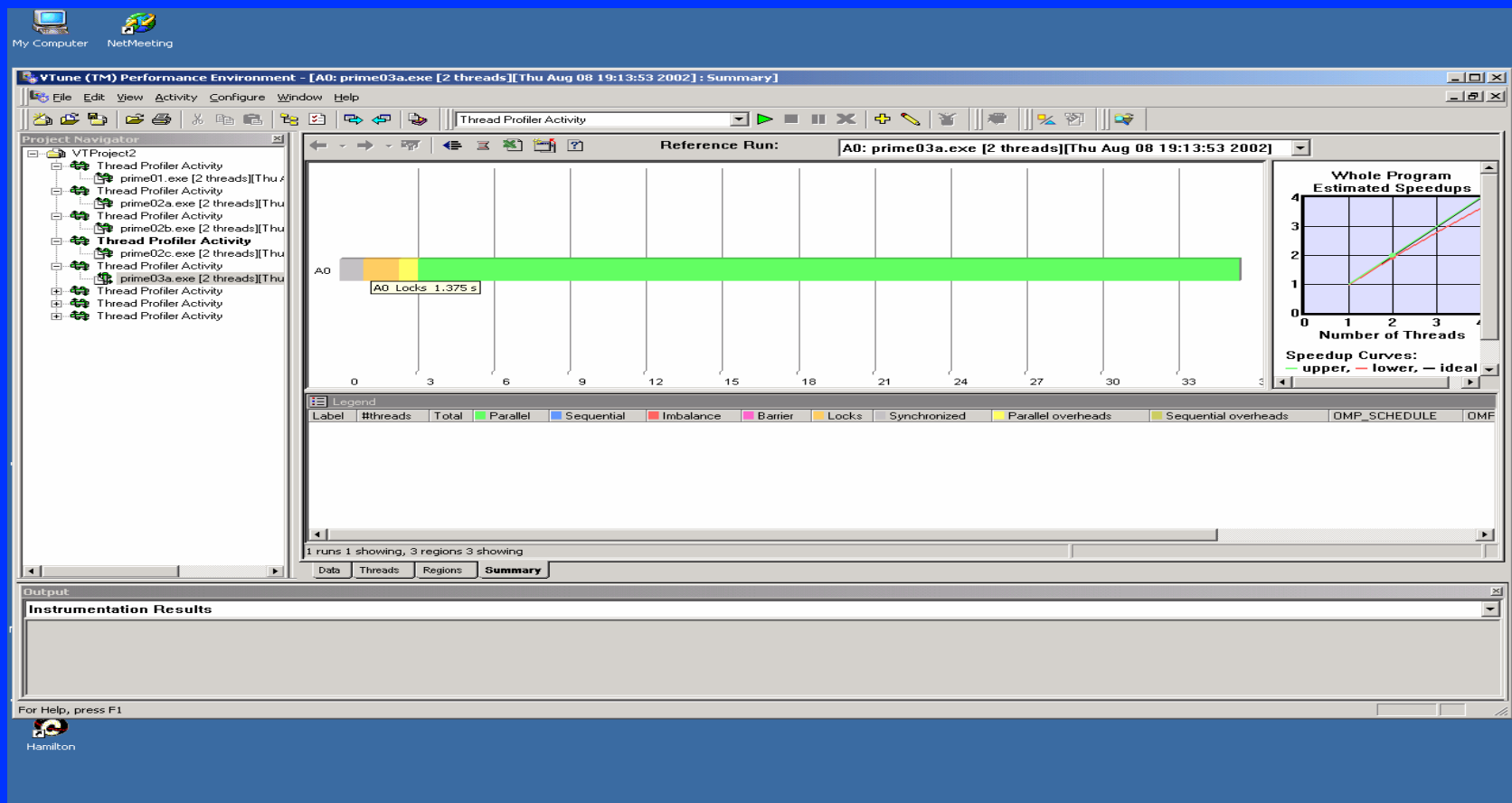
Let us try Another Scheduling Technique



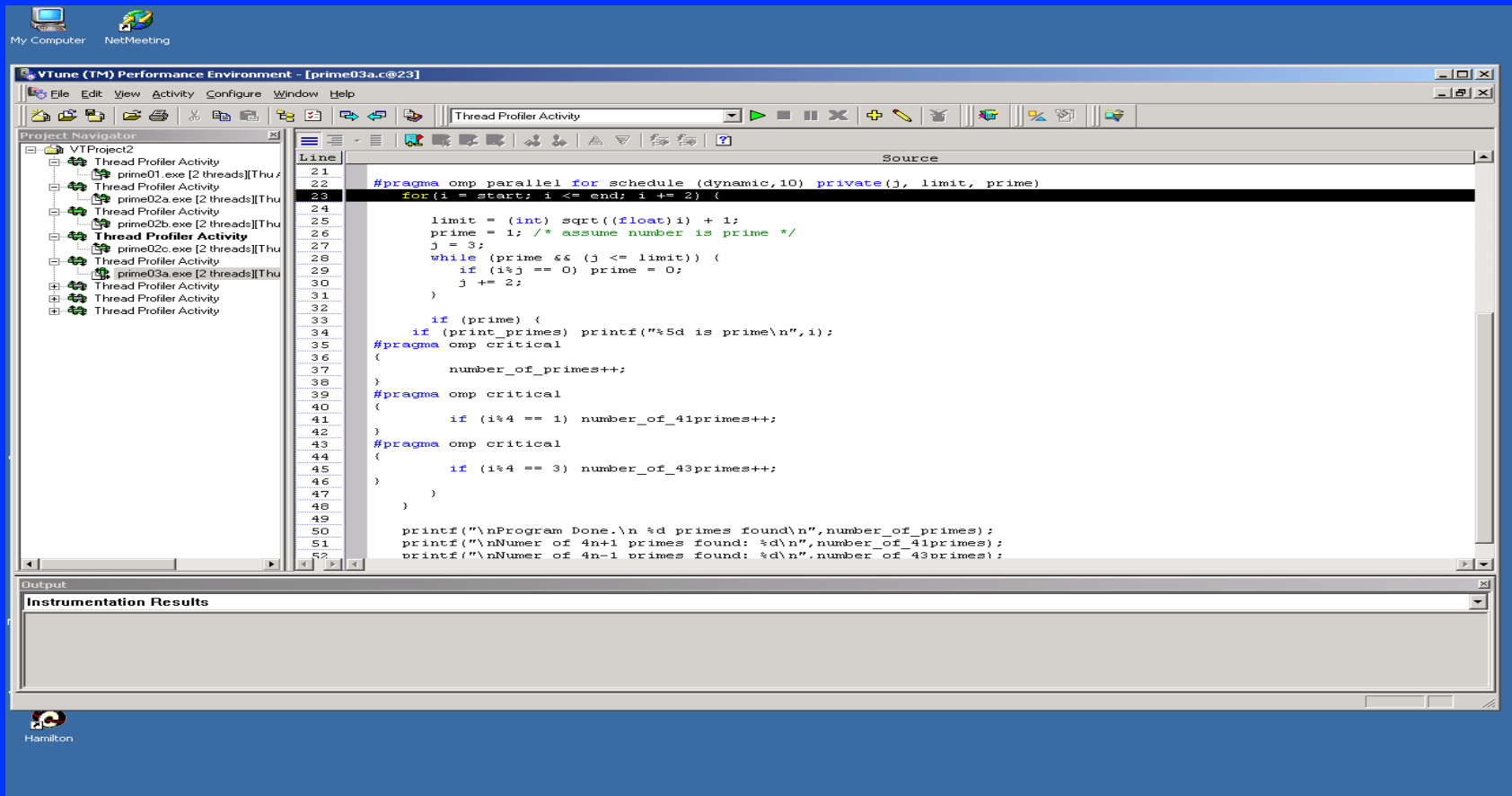
Result of Guided Scheduling



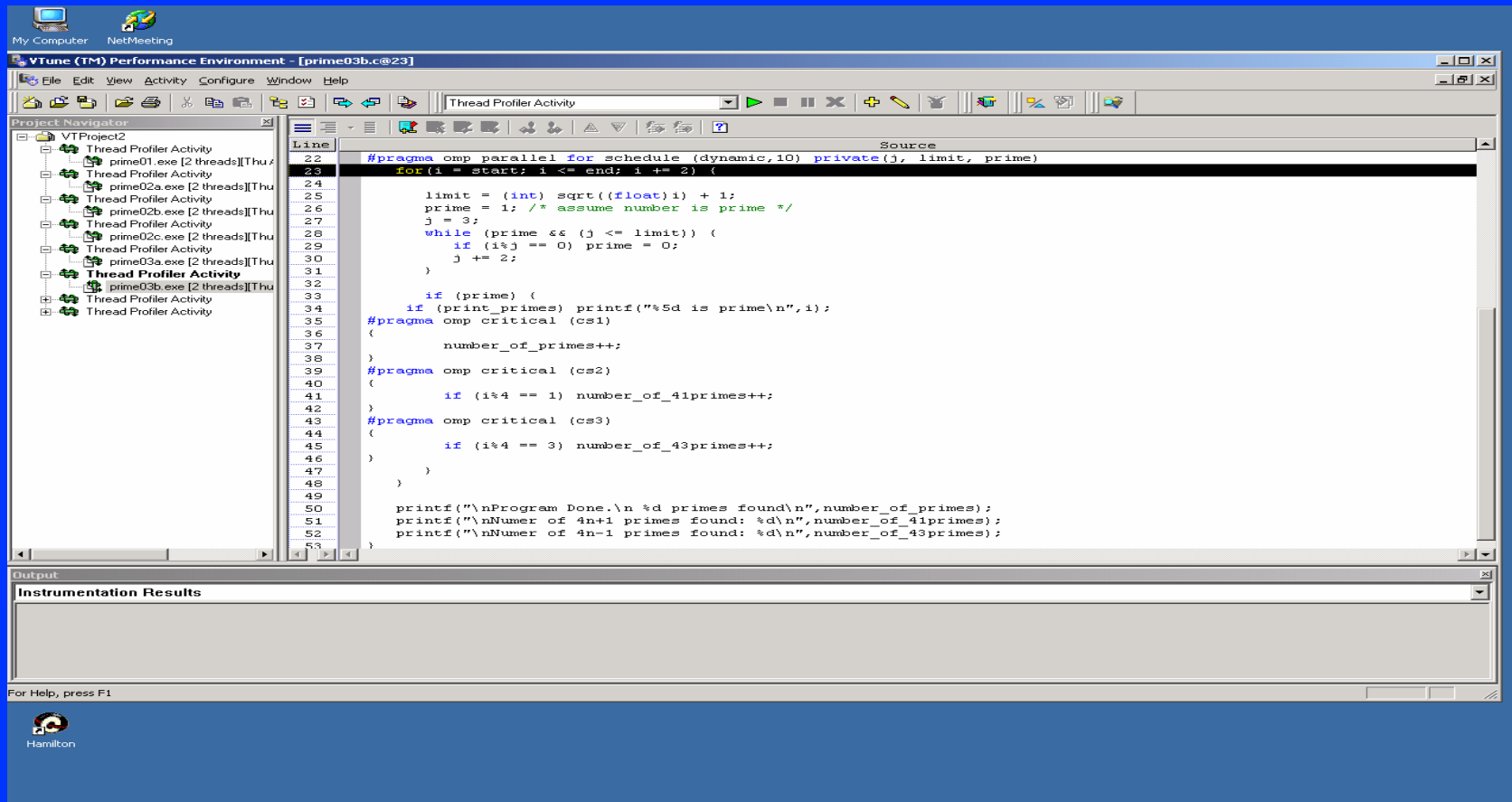
Display of Execution Time Inside Locks



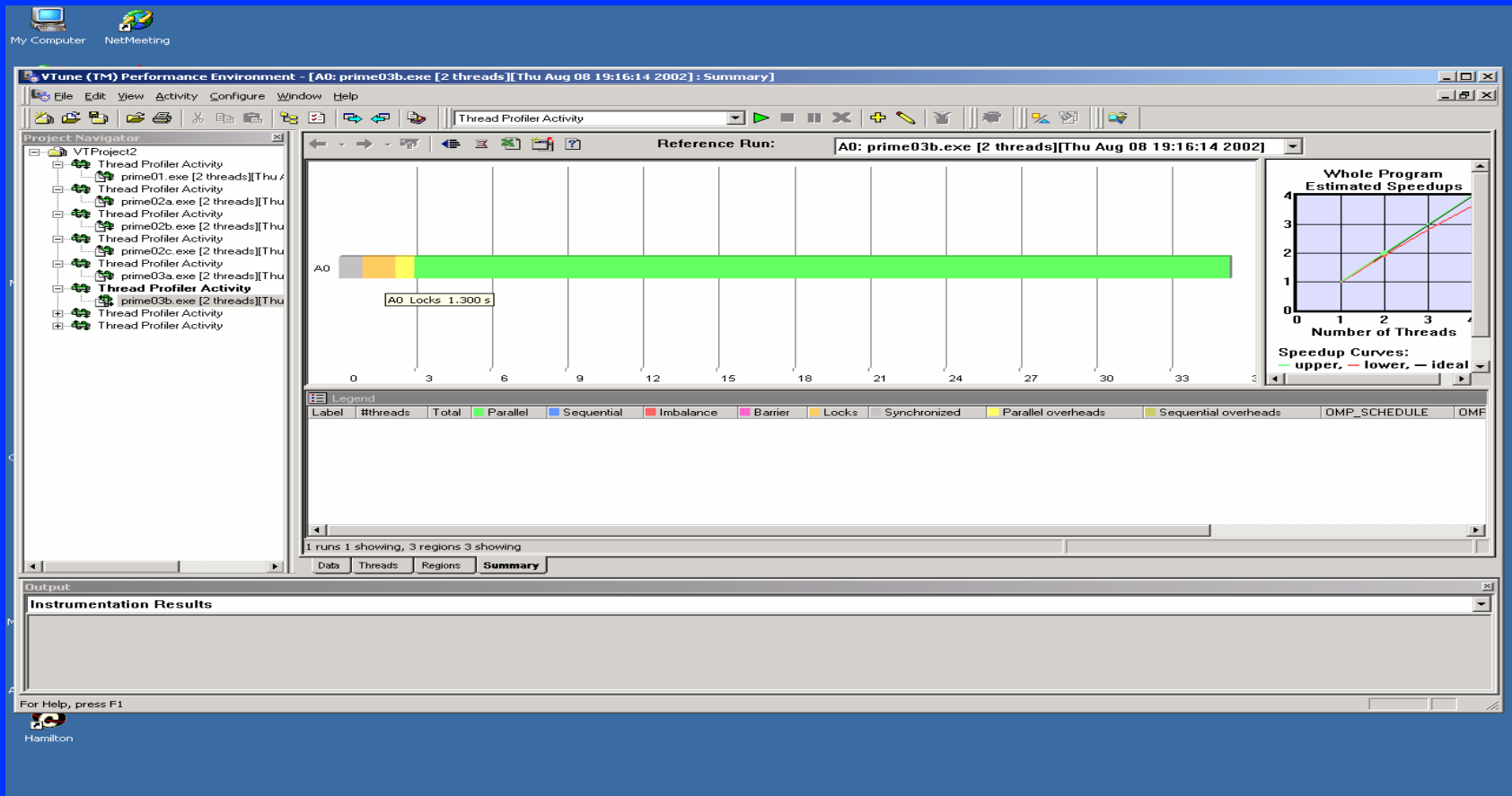
Notice the Three Critical Sections



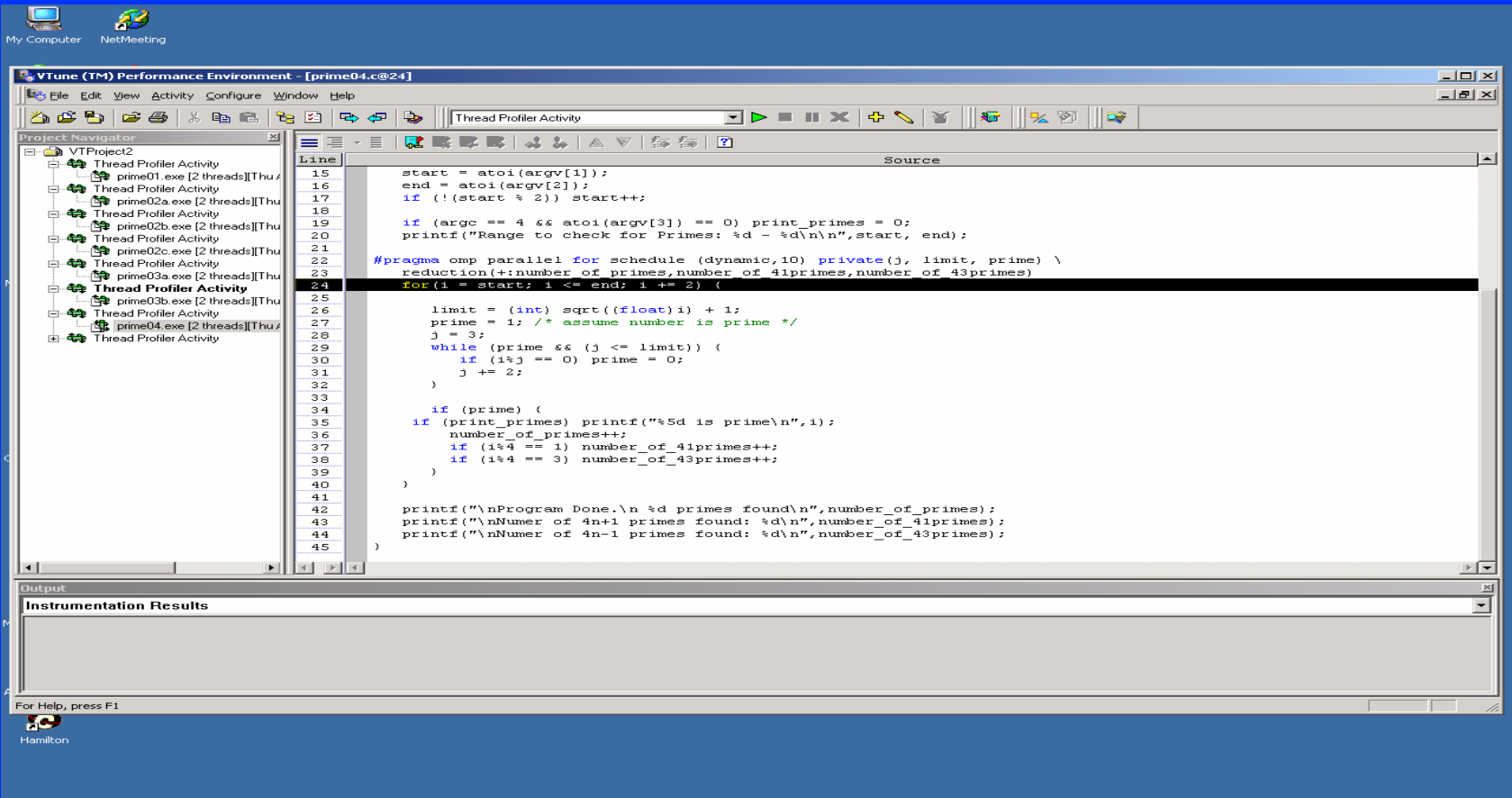
What About Named Critical Sections?



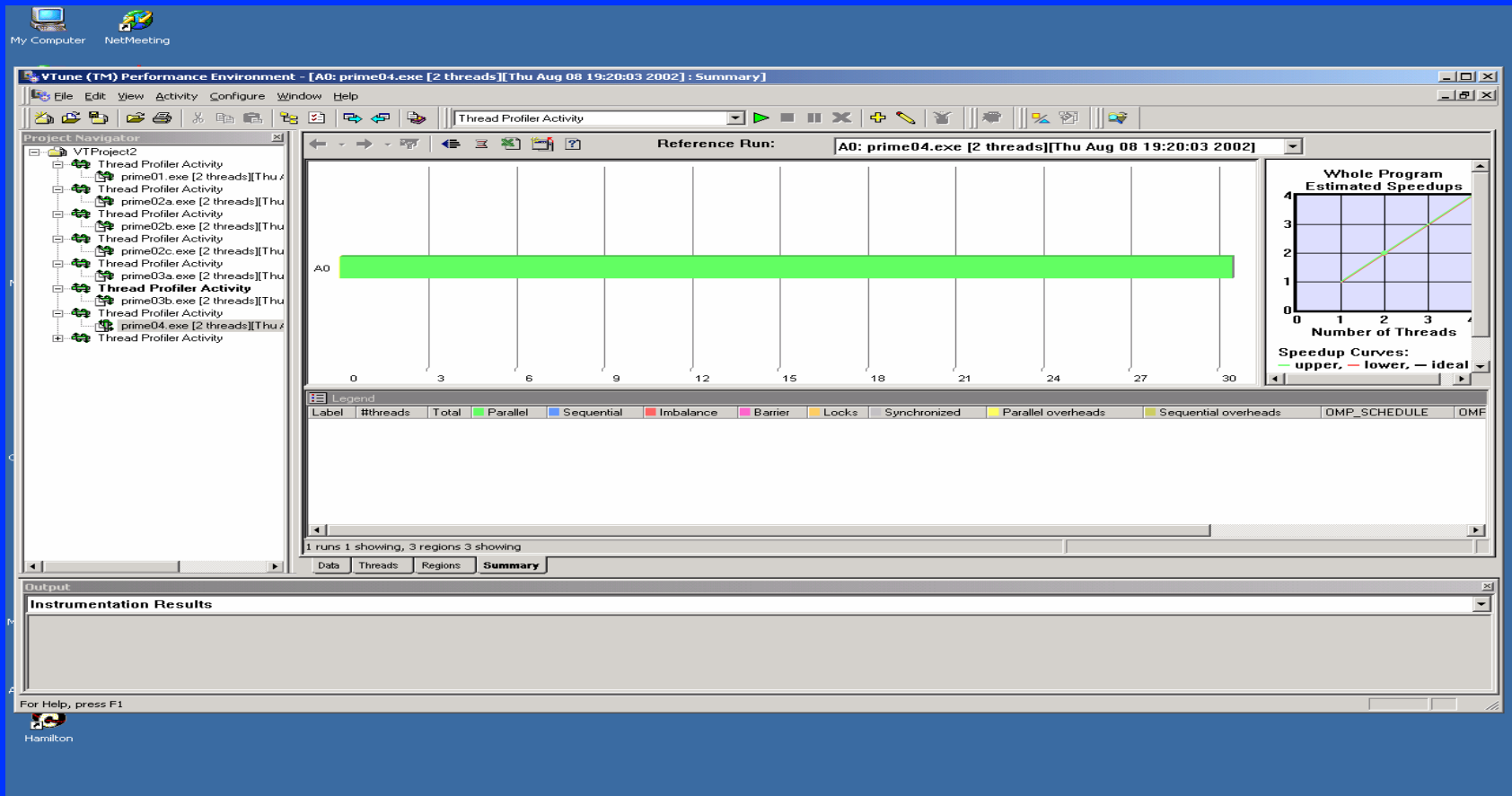
Execution Time for Named Critical Sections



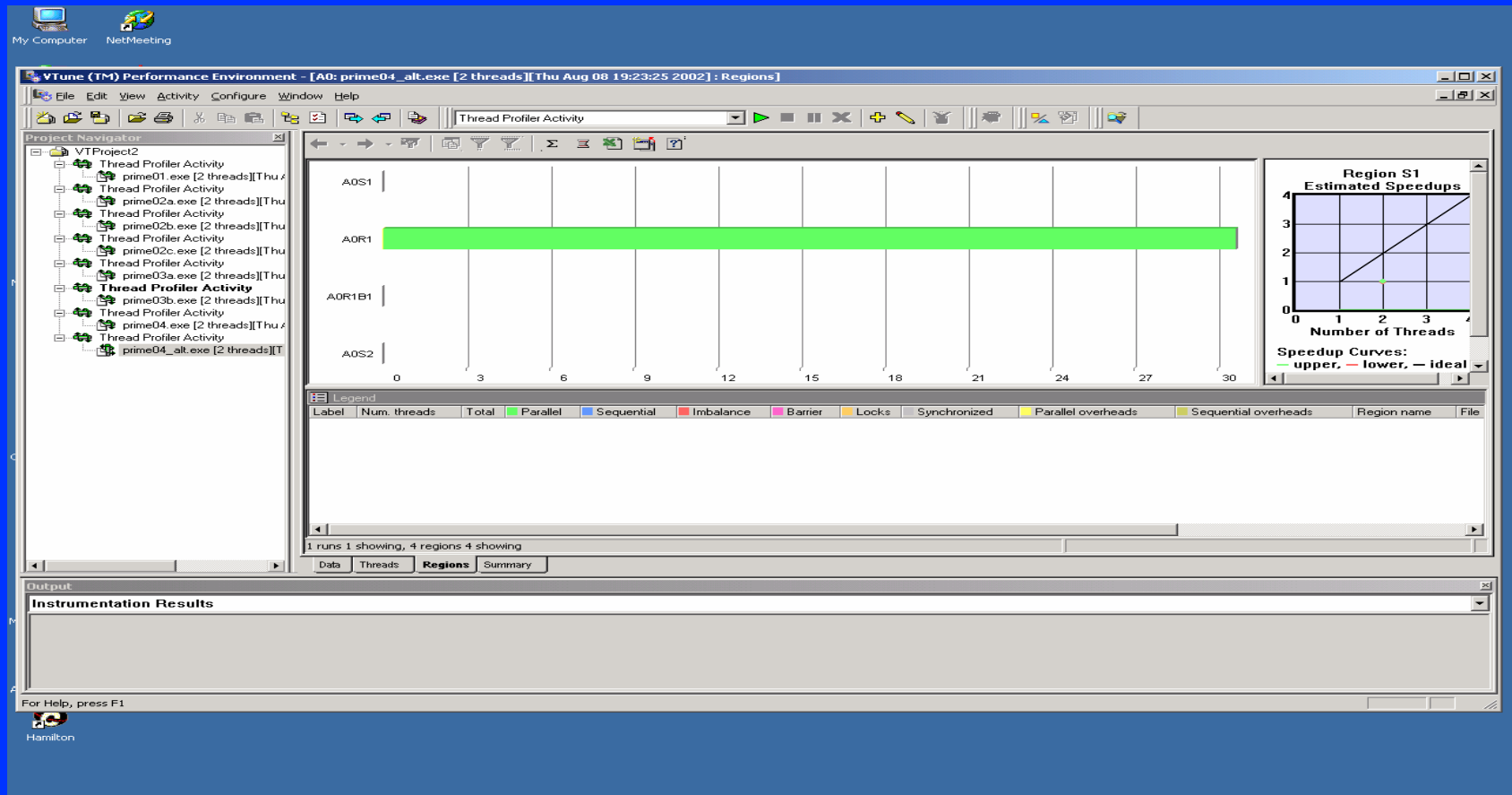
What if We Replace the OpenMP Critical Sections with an OpenMP Reduction Clause?



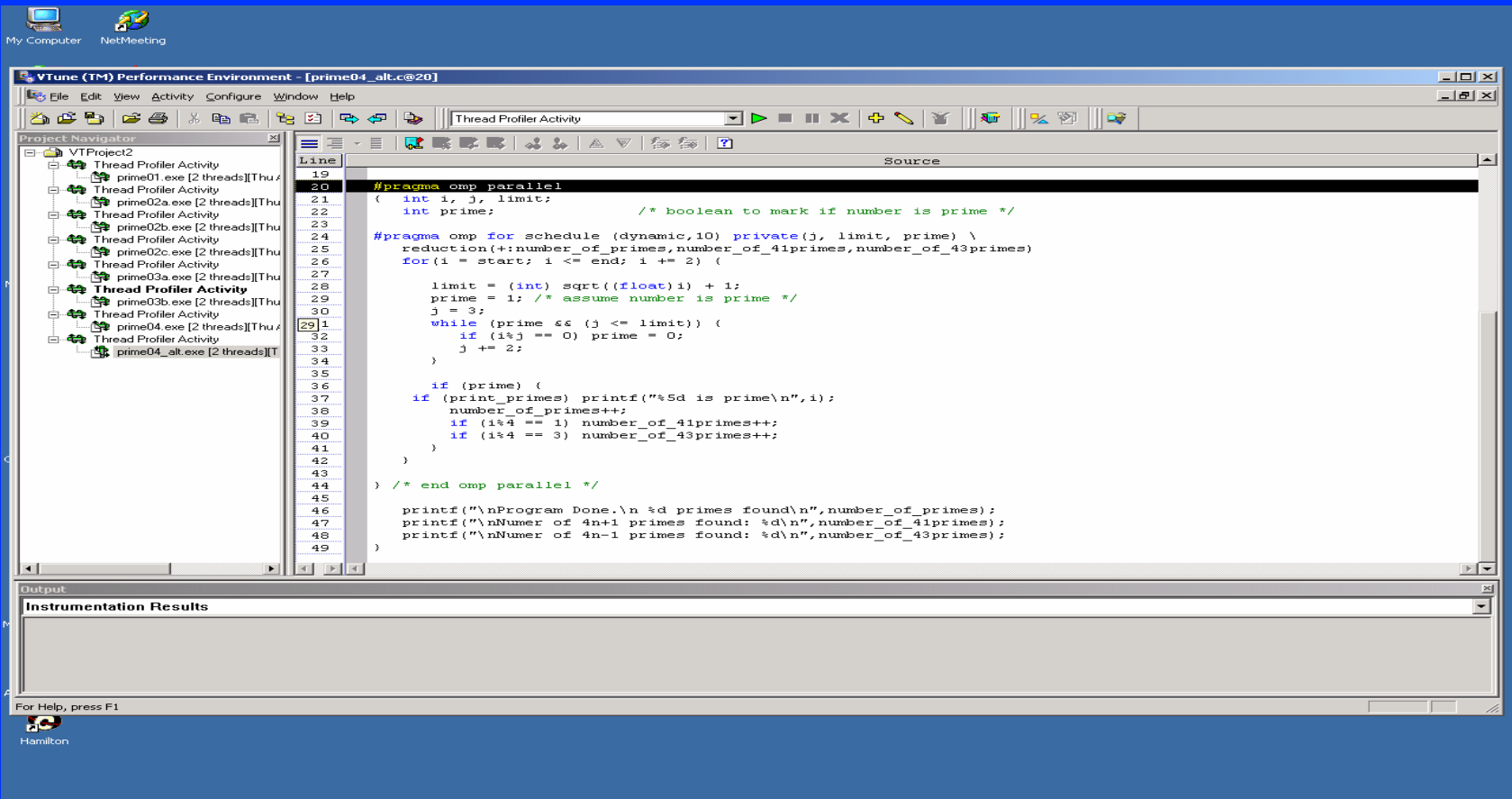
Execution Profile as a Result of Using the Reduction Clause



Alternative Solution with Reduction Clause



Alternative Solution with a Dynamic Chunk Size Specification



Some General Information about Sieves and OpenMP

- A sieve is an exhaustive search technique
- Sieves try to eliminate non-solutions instead of trying to find solutions
- Straight forward application of sieves will result in algorithms whose time requirements are prohibitive
- To be useful, the implementer must use a sieve technique as a *framework* within which to approach the problem
- OpenMP and/or other parallel programming techniques are not necessarily panaceas for exhaustive search problems

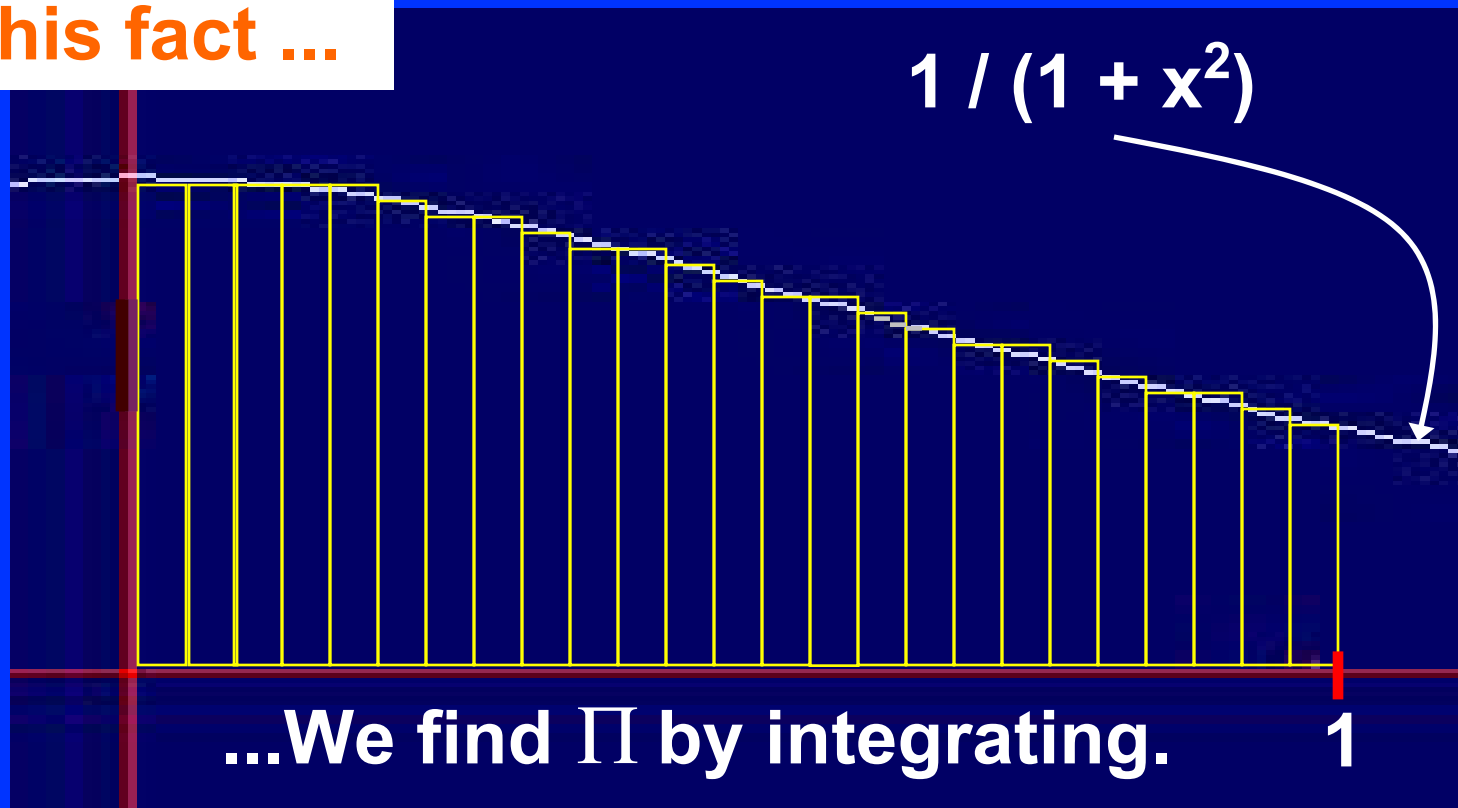
How Does the Intel® Thread Checker Work?

Let us look at an initial draft of an OpenMP parallel programming application and then proceed to correct execution errors using the Intel® Thread Checker within VTune™

Calculating Π

$$\int_0^1 \frac{1}{(1+x^2)} dx = \frac{\pi}{4}$$

Using this fact ...



Π Program

```
static long num_steps = 100000;
double step;
void main ()
{
    int i;  double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for (i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

C/C++ parallel for Pragma

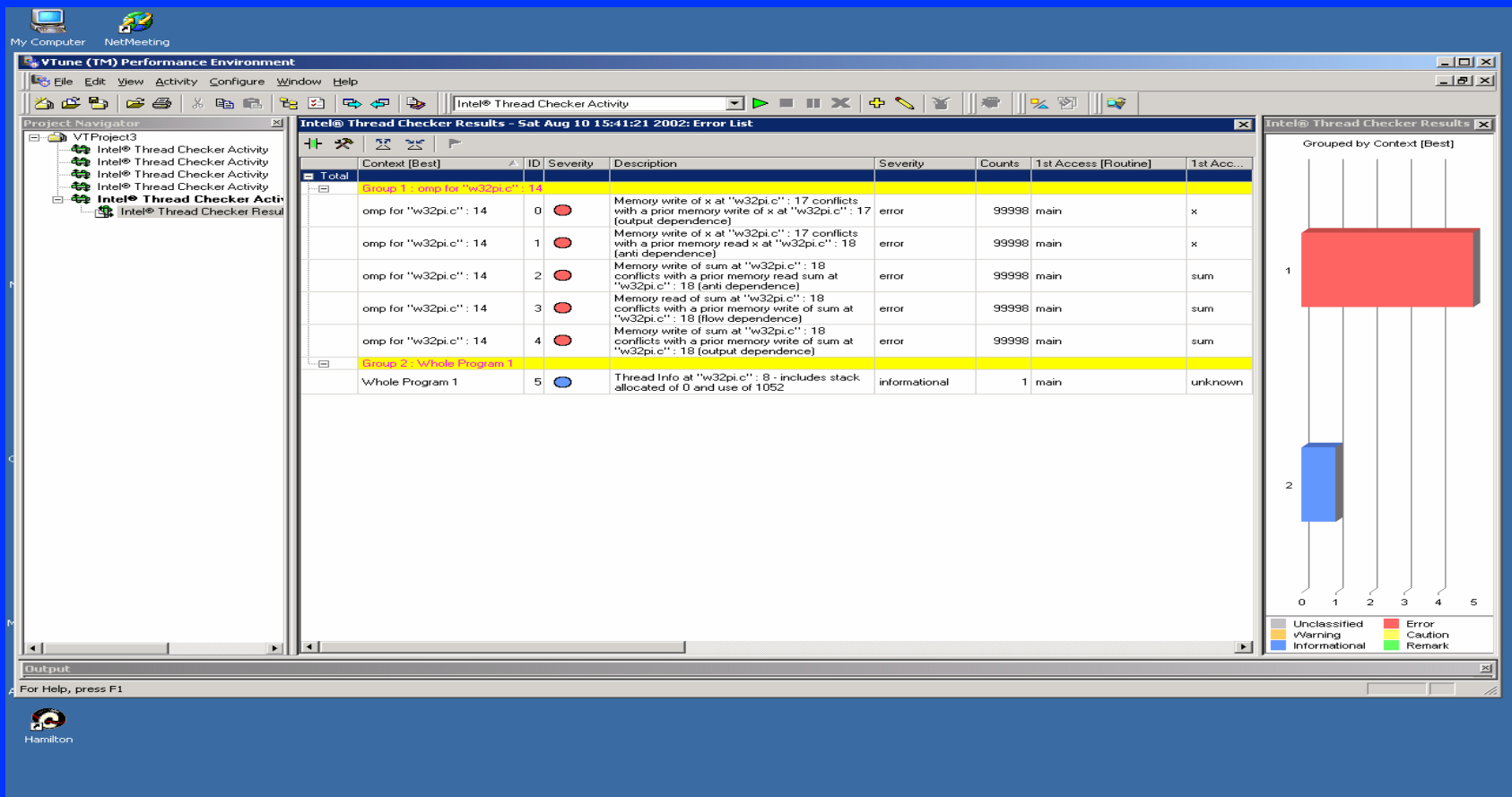
```
#pragma omp parallel for  
for (i=0; i < N; ++i) {  
    disjoint_computation(i);  
}
```

**Threads cooperate
to do
disjoint iterations
of the loop.**

```
double x, sum = 0.0;  
#pragma omp parallel for  
for (i=1; i<= num_steps; i++) {  
    x = (i-0.5)*step;  
    sum = sum + 4.0/(1.0+x*x);  
}
```

**But, this loop
has data races.**

Diagnostics Generated by the Thread Checker



Associating Error Messages with the User's Source

The screenshot shows the Intel Thread Checker Activity window in the VTune Performance Environment. The window is divided into several panes:

- Project Navigator:** Shows the project structure with 'VTProject3' and 'Intel Thread Checker Activity'.
- Intel Thread Checker Results - Sat Aug 10 15:41:21 2002: Error List:** A table listing error messages. The table has columns: Context (Best), ID, Severity, Description, Severity, Counts, 1st Access [Routine], and 1st A....
- Stack Trace:** Shows the stack trace for the selected error message, including the function 'omp for main "w32pi.c": 14'.
- Source:** Displays the source code for the selected error message, showing a parallel loop in 'w32pi.c'.
- Intel Thread Checker Results - 5:** A bar chart showing the results of the thread checker, grouped by context (Best).

The error list table contains the following data:

Context (Best)	ID	Severity	Description	Severity	Counts	1st Access [Routine]	1st A...
Group 1 - omp for "w32pi.c": 14							
omp for "w32pi.c": 14	0	Error	Memory write of x at "w32pi.c": 17 conflicts with a prior memory write of x at "w32pi.c": 17 (output dependence)	error	99998	main	x
omp for "w32pi.c": 14	1	Error	Memory write of x at "w32pi.c": 17 conflicts with a prior memory read x at "w32pi.c": 18 (anti dependence)	error	99998	main	x
omp for "w32pi.c": 14	2	Error	Memory write of sum at "w32pi.c": 18 conflicts with a prior memory read sum at "w32pi.c": 18 (anti dependence)	error	99998	main	sum
omp for "w32pi.c": 14	3	Error	Memory read of sum at "w32pi.c": 18 conflicts with a prior memory write of sum at "w32pi.c": 18 (flow dependence)	error	99998	main	sum
			Memory write of sum at "w32pi.c": 18				

The stack trace shows the following context:

Stack Trace: omp for main "w32pi.c": 14

The source code shows the following lines:

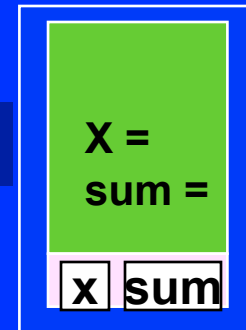
```
1 static long num_steps = 100000;
2 double step;
3 void main()
4 {
5     int i;
6     double x, pi, sum = 0.0;
7     step = 1.0 / (double) num_steps;
8     #pragma omp parallel for
9     for (i = 1; i < num_steps; i++)
10     {
11         x = (i - 0.5) * step;
12         sum = sum + 4.0 / (1.0 + x * x);
13     }
14     pi = step * sum;
```

C/C++ parallel for Clauses

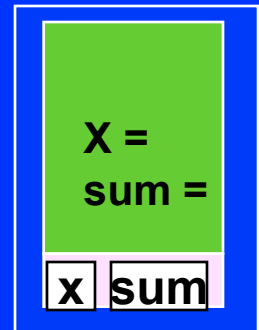
```
#pragma omp parallel for private(x,y,z)
```

```
#pragma omp parallel for reduction(+: sum)
```

Thread 0



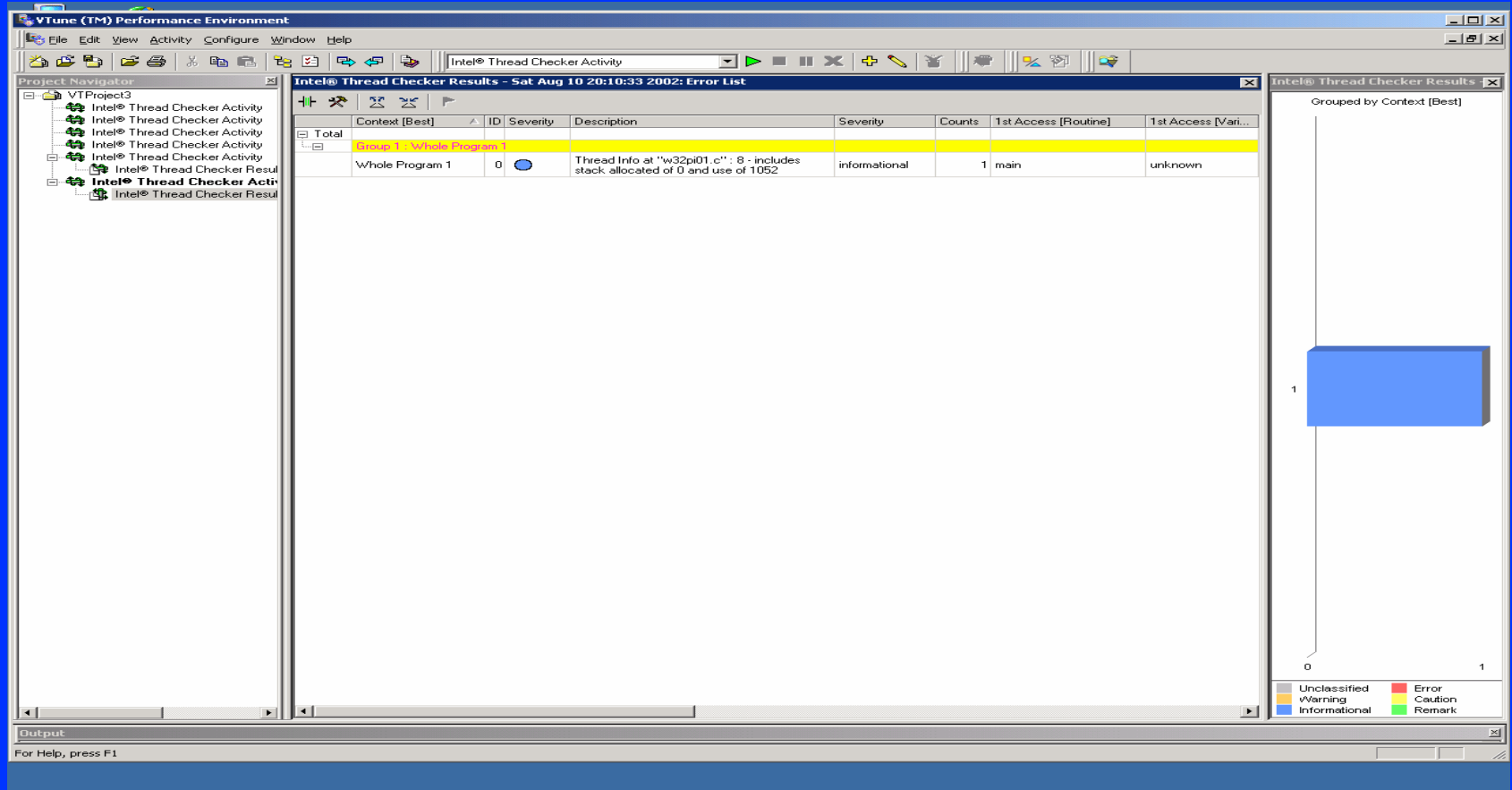
Thread 1



□ `x` □ `sum`

```
double x, sum = 0.0;
#pragma omp parallel for private(x) reduction(+:sum)
for (i=1; i<= num_steps; i++) {
    x = (i-0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
}
```

Intel® Thread Checker Analysis after Updating the OpenMP Pragma Semantics



Examination of Intel® Thread Checker Diagnostics and User's Source Code

The screenshot displays the VTune (TM) Performance Environment interface. The main window is titled "Intel® Thread Checker Results - Sat Aug 10 20:10:33 2002: Error List". It shows a table with columns: Context [Best], ID, Severity, Description, Severity, Counts, 1st Access [Routine], and 1st Access [Vari...]. The table lists results for "Whole Program 1" with ID 0, severity informational, and 1 count. The description states: "Thread info at 'w32pi01.c': 8 - includes stack allocated of 0 and use of 1052".

Below the table, the "Intel® Thread Checker Results - Sat Aug 10 20:10:33 2002 (ID=0) Definition" window is open, showing the source code for "main 'w32pi01.c': 8". The code is as follows:

```
1 static long num_steps = 100000;
2
3 double step;
4
5 void main()
6 {
7     int i;
8     double x, pi, sum = 0.0;
9     step = 1.0 / (double) num_steps;
10     #pragma omp parallel for private(x) reduction(+:sum)
11     for (i = 1; i < num_steps; i++)
12     {
13         x = (i - 0.5) * step;
14         sum = sum + 4.0 / (1.0 + x * x);
15     }
16     pi = step * sum;
17 }
```

The "Stack Trace" window shows the call stack for the selected thread, with the top entry being "main 'w32pi01.c': 8". The "Output" window at the bottom shows the message "For Help, press F1".

The Intel® Thread Checker “Automatic” Debugger:

- Finds parallel bugs:
 - Data races
 - Uninitialized variables
 - Failure to copy private → shared
- Uses computer time, not human time

Conclusions

- Intel Thread Analyzer helps you tune your parallel programming application
- Intel® Thread Checker helps you debug your parallel programming application

Other names and brands may be claimed as the property of others.